

---

# **Serializer Documentation**

***Release 0.1***

**Konsta Vesterinen**

February 19, 2013



# CONTENTS

<b>1</b>	<b>QuickStart</b>	<b>3</b>
<b>2</b>	<b>How to handle relations</b>	<b>5</b>
<b>3</b>	<b>Using exclude, include and only</b>	<b>7</b>
<b>4</b>	<b>Using attribute sets</b>	<b>9</b>
4.1	API Documentation . . . . .	9
4.2	Changelog . . . . .	12
4.3	License . . . . .	12
	<b>Python Module Index</b>	<b>15</b>



Serializer provides Easy object serialization. Mimics RoR ActiveRecord serializer.



# QUICKSTART

1. Make your class inherit Serializable mixin.
2. Define attributes() method and make it return a list of property names

Example:

```
from serializer import Serializable

class User(Serializable):
    first_name = 'John'
    last_name = 'Matrix'
    email = 'john.matrix@example.com'
    protected_property = 'this value is not included in json'

    def attributes(self):
        return ['first_name', 'last_name', 'email']

user = User()
user.as_json()
# '{"first_name": "John", "last_name": "Matrix"}'

user.as_json_dict()
# {'first_name': 'John', 'last_name': 'Matrix'}
```





# HOW TO HANDLE RELATIONS

Serializer supports serializing of deep object structures. Let's say we have two classes `Event` and `Location` and we want to jsonify an event along with its location.

```
class Location(Serializable):
    name = 'Some Location'

    def attributes(self):
        return ['name']

class Event(Serializable):
    name = 'Some Event'
    location = Location()

    def attributes(self):
        return ['name', 'location']

event = Event()
event.as_json()
# '{"name": "Some Event", "location": {"name": "Some Location"}}'
```



## USING EXCLUDE, INCLUDE AND ONLY

You can fine-grain what gets included in json format by using `exclude`, `include` and `only` parameters.

```
User.as_json(only=['name', 'email']) # include only name and email
```

```
# include all the fields defined in attributes as well as age
User.as_json(include=['age'])
```

```
# include all the field defined in attributes but exclude 'email'
User.as_json(include=['email'])
```



# USING ATTRIBUTE SETS

Many times you may have situations where having one default attribute list is not enough. For example you may have multiple views that return user details and many views that return user with only its basic info.

```
class User(Serializable):
    first_name = 'John'
    last_name = 'Matrix'
    email = 'john.matrix@example.com'
    age = 33
    is_active = True

    def attributes(self):
        return ['first_name', 'last_name', 'email']

    def attribute_sets(self):
        return {'details': ['age', 'is_active']}
```

```
user = User()
user.as_json_dict(include='details')
'''
{
  'first_name': 'John',
  'last_name': 'Matrix',
  'email': 'john.matrix@example.com',
  'age': 33,
  'is_active': True
}
'''
```

## 4.1 API Documentation

**class** `serializer.Serializable`

This class mimics the functionality found in RoR ActiveRecord. For more info see:  
<http://api.rubyonrails.org/classes/ActiveModel/Serializers/JSON.html>

**as\_json** (*only=None, exclude=None, include=None*)

Returns object attributes as a dictionary with jsonified values

### Parameters

- **only** – a list containing attribute names to only include in the returning dictionary
- **exclude** – a list containing attributes names to exclude from the returning dictionary

- **include** – a list containing attribute names to include in the returning dictionary

Without any options, the returned JSON string will include all the fields returned by the models attribute() method. For example:

```
>>> class User(Serializable):
...     def attributes(self):
...         return [
...             'id',
...             'first_name',
...             'last_name'
...         ]
...
>>> user = User()
>>> user.first_name = 'John'
>>> user.last_name = 'Matrix'
>>> user.as_json()
{"id": 1, "first_name": "John", "last_name": "Matrix"}
```

The 'only' and 'exclude' options can be used to limit the attributes included. For example:

```
>>> user.as_json(only=['id', 'first_name'])
{"id": 1, "first_name": "John"}

>>> user.as_json(exclude=['id'])
{"first_name": "John", "last_name": "Matrix"}
```

To include the result of some additional attributes, method calls, attribute sets or associations on the model use 'include'.

```
>>> user.weight = 120
>>> user.as_json(include=['weight'])
{"id": 1, "first_name": "John", "last_name": "Matrix", "weight": 120}
```

Sometimes its useful to assign aliases for attributes. This can be achieved using keyword 'as'.

```
>>> user.as_json(only=['first_name as alias'])
{"alias": "John"}
```

In order to fine grain what gets included in associations you can use 'include' parameter with additional arguments.

```
>>> user.as_json(include=[('posts', {'include': 'details'})])
{
  "id": 1,
  "name": "John Matrix",
  "first_name": "John",
  "last_name": "Matrix",
  "weight": 100,
  "posts": [
    {"id": 1, "author_id": 1, "title": "First post"},
    {"id": 2, "author_id": 1, "title": "Second post"}
  ]
}
```

Second level and higher order associations work as well:

```
>>> user.as_json('include'= [('posts',
...     {
...         'include': [
...             ('comments', {'only': ['body']})
...         ]
...     })])
```

```

...         ],
...         'only': ['title']
...     }
... )]
{
    "id": 1,
    "first_name": "John",
    "last_name": "Matrix",
    "weight": 100,
    "posts": [
        {
            "comments": [
                {"body": "1st post!"}, {"body": "Second!"}
            ],
            "title": "Welcome to the weblog"
        },
    ]
}

```

#### **attribute\_sets()**

This method should return a dict with keys as attribute set names and values as desired attribute sets

Attribute sets can be used as a convenient shortcuts in `as_json()`

Examples:

```

>>> User(Serializable):
...     def attribute_sets(self):
...         return {'details': ['id', 'name', 'age']}
>>> User(id=1, name='someone',).as_json(only='details')
{'id': 1, 'name': 'Someone', 'age': 14}

```

#### **attributes()**

This method is being used by `as_json` for defining the default json attributes for this model.

Serializable objects can override this and return a list of desired default attributes.

Examples:

```

>>> User(Serializable):
...     def __init__(name, age):
...         self.name = name
...         self.age = age
...
...     def attributes(self):
...         return ['name', 'age']
...
>>> User('John', 50).as_json()
{'name': 'John', 'age': 50}

```

#### **to\_json (only=None, exclude=None, include=None)**

Returns the object attributes serialized in json format

##### **Parameters**

- **only** – a list containing attribute names to only include in the returning json
- **exclude** – a list containing attributes names to exclude from the returning json
- **include** – a list containing attribute names to include in the returning json

**to\_xml** (*only=None, exclude=None, include=None, \*\*kwargs*)

Returns the object attributes serialized in xml format

#### Parameters

- **only** – a list containing attribute names to only include in the returning xml
- **exclude** – a list containing attributes names to exclude from the returning xml
- **include** – a list containing attribute names to include in the returning xml

**serializer.register\_dumper** (*key, dumper\_callable*)

Registers new dumper for given class type

#### Examples::

```
>>> class MyClassA(object):
...     pass
>>> class MyClassB(MyClassA):
...     pass
>>> register_dumper('MyClassA', lambda a: 'myclass')
>>> dump_object(MyClassA())
"myclass"
>>> dump_object(MyClassB())
<MyClassB instance>

>>> class MyClassA(object):
...     pass
>>> class MyClassB(MyClassA):
...     pass
>>> register_dumper(MyClassA, lambda a: 'myclass')
>>> dump_object(MyClassA())
"myclass"
>>> dump_object(MyClassB())
"myclass"
```

## 4.2 Changelog

Here you can see the full list of changes between each Serializer release.

### 4.2.1 0.2.1 (2013-02-16)

- Added API documentation

### 4.2.2 0.2 (2013-01-26)

- Added XML serialization

### 4.2.3 0.1 (2012-12-04)

- Initial public release



## 4.3 License

Copyright (c) 2012, Konsta Vesterinen

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- The names of the contributors may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.



# PYTHON MODULE INDEX

## S

serializer, 9